Design Specification for JTF Page Fabrication, CRM Applications, 11*i*

Project ID:<project ID#/project name>

[redacted]

Status: <Review Draft/Approved>

[redacted]

URL for this version is: **http://crm.us.oracle.com/**

Change Log		
Version	**Reviewers**	**Changes**
[redacted]		

Review and Approval

The reviewers should have the technical knowledge and authority to state that the project is doing the right thing. If they do not have the authority and knowledge to do this, then they should not be on a review team. (This restriction is for formal technical reviews that determine acceptance. The author is always free to call for informal peer reviews to obtain guidance without approval.) Recommended reviewers are specified by the Product Manager.

A formal technical review ends with accepting or not accepting the reviewed product. The review team accepts the document when they believe it is right. Right means the product has selected the correct alternatives, is complete, and is worth doing (satisfies requirements). Update the title page "Status" line to *<Approved>* when all issues are resolved and the document is ready for source control.

Table of Contents

1 Introduction

JTF Page Fabrication service speeds up page delivery for JSP applications.

It is assumed that readers of this Design Specification have read and are familiar with the contents of the following documents:

- *<Project Name> Functional Specification*, Version x.y([<Project Name> FS])

2 High Level Design

The key idea behind page prefabrication is to create and store the HTTP Response for a URL request a priori, so that when an user actually requests the URL it can be serviced immediately. The response is stored on the file system. Following diagrams show two different deployment strategies for the prefabricator. Figure 1 shows the case when the prefabricator sits in the Apache_Jserv[1] layer, while Figure 2 shows the case when the prefabricator communicates with Calypso[2]. In the former case, the prefabricator requires to both issue prefetch requests, and actually manage the storage and retrieval of the prefetched response. In the latter case, the prefabricator is primarily concerned with issuing prefetch requests and Calypso handles the cache management.

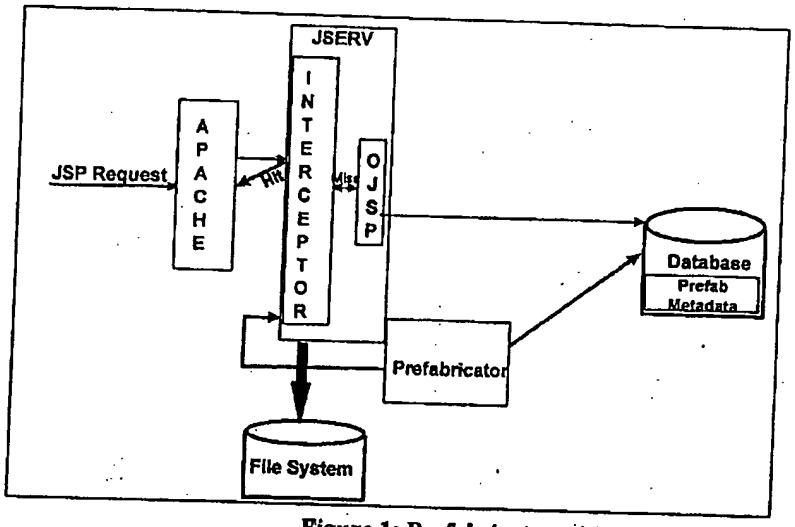
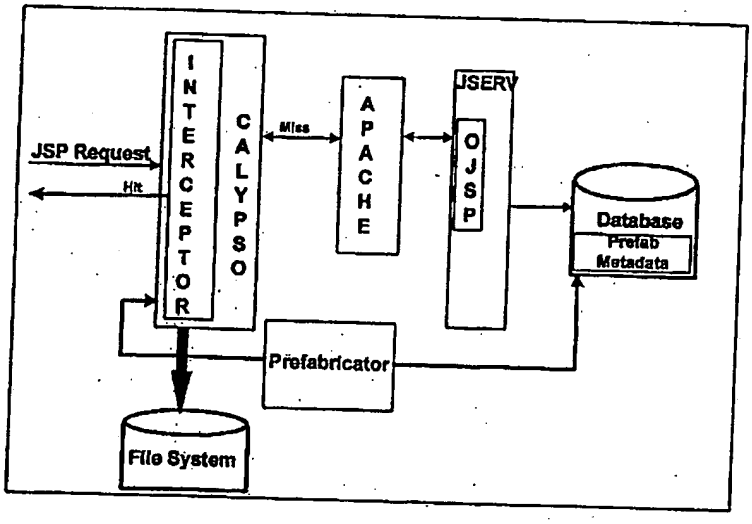


Figure 1: Prefabricator with Jserv



[1] More documentation on Apache can be found at: http://www.apache.org
[2] Calypso is a reverse-proxy server that sits between browser clients and the Apache server. More documentation can be found at: http://calypso.oracle.com

Figure 2: Prefabricator with Calypso

Design

The three main components:

Prefabricator: this intelligently collects/generates all the URLs to be prefabricated, and issues the URL requests to get it prefabricated. Furthermore, this component needs to ensure consistency of the prefabricated data. More details about this component follow.

Interceptor: this intercepts the URL requests, determines if the request can be serviced from the cache or not. Furthermore, the interceptor needs to distinguish between "client" URL requests and the "prefabricator" URL requests.

Cache: this is responsible for storage and retrieval of prefetched responses

The primary focus of this project is to build the prefabricator and interceptor technology. Existing caching technologies (such as Calypso) will be leveraged for the cache component. Following subsections explain the above three components in more details.

3 Design Description

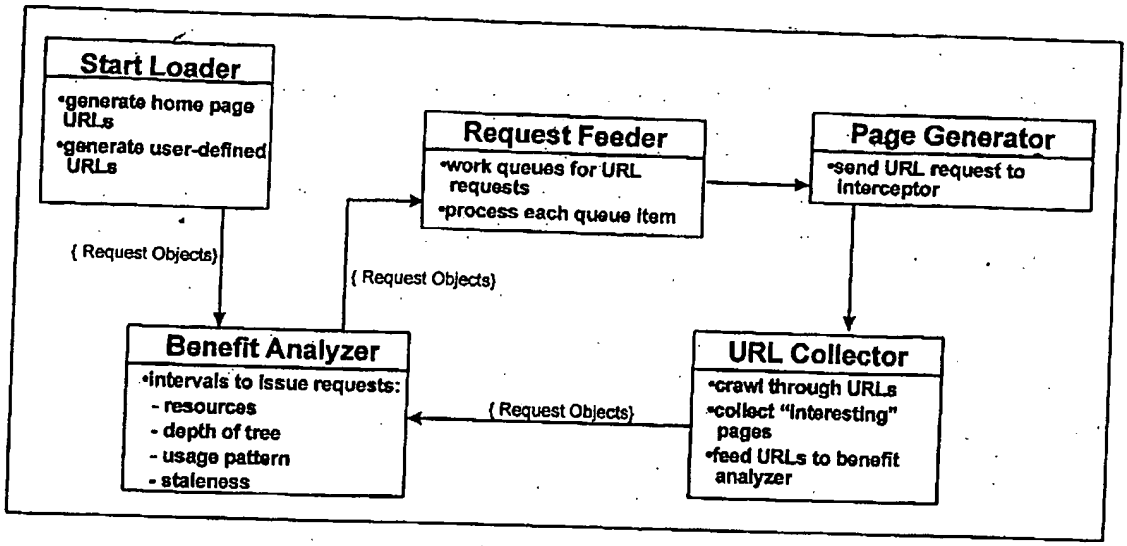


Figure 3: Prefabricator Design

Figure 3 above shows the flow between the different components of the Prefabricator. The basic unit of work done by the prefabricator is fabricating a URL: i.e., issuing a request to the web-server (or interceptor) to get the response generated. This section provides design details about each of the components in the prefabricator:

Common Data Structures:

- PageRequestBlock class
- PolicyInformation class
- SystemStatistics class

Components

- StartLoader
- BenefitAnalyzer
- RequestFeeder
- PageGenerator
- URLCollector
- Invalidator

3.1 Prefabricator Data Structures

3.1.1 Page Request Block (PRB)

Every URL that is a candidate for prefabrication is represented as a page request block.

This is the basic data structure that gets passed around the system. It encapsulates the following information:

Attribute	Data Type	Description
URI	String	URI that the request represents
userID	int	userID of the user this request belongs to
appID	int	appID this request belongs to
respID	int	respID of the user this request belongs to
depth	int	starting from the home page, the number of pages to be navigated before this URL is reached. A page is at depth 0 if it is the user's homepage, depth 1 if it is a page that can be reached directly from the home page, etc. Login page is at depth -1.
Weight	long	weight associated with this PRB
homePRB	PRB	PRB of the entry page to this application for the user
fileLocator	String	location of the fabricated page on the file system
requestTimeStamp	Date	time stamp when request block was created
responseTimeStamp	Date	time stamp when page was fabricated

All of the above mentioned attributes will have *get* and *set* methods. Following is the list of constructors for this object:

1. PageRequestBlock(String URI, int depth): assumes that this PRB is the first PRB
2. PageRequestBlock(String URI, int depth, PageRequestBlock originator): assumes this PRB inherits userID, appID and respID values from originator
3. PageRequestBlock(String URI, int userID, int appID, int respID, int depth, PageRequestBlock home)

Who construct PRBs?

1. startLoader constructs entry point PRBs (uses constructor 1)
2. URLCollector constructs PRBs from a given PRB response stream (uses constructor 2)
3. BenefitAnalyzer modifies PRBs to set the weight and requestTimeStamp attributes
4. PageGenerator modifies PRBs to set the responseTimeStamp attributes

3.1.2 Policy Table (PT)

The Policy Table stores information to tune the prefabricator system.

Administrators will configure the policies using the Admin Console. The policies are striped by

application and persisted in the database. When the prefabricator is started up, the policy table is loaded up in memory - each row in the policy table corresponds to an instance of the PolicyObject.

The policies supported by the policy table are:

- Depth: defines how deep down from the home page is candidate for fabrication.
- CPU consumption limit: the maximum CPU power available for the prefabricator
- Responsibilities: define for which responsibilities we should prefabricate
- Refresh interval: defines how often a page should be re-fabricated

The following table is an example of the policy table

App ID	Depth	CPU consumption limit	Responsibility	Refresh interval (sec)
690	2	60%	Managers, Representatives	3600
670	1	60%	Managers	10800

The PolicyInformation Class models each row of the policy table. This class will take care of persistency, loading and saving to database. The following accessor methods will be used:

public int getDepth()

public void setDepth(int depth)

public int getCpuConsumptionLimit()

public void setCpuConsumptionLimit(int cpuConsumptionLimit)

public Vector getResponsibilities()

public boolean containsResponsibility(Integer resp)

public void addResponsibility(Integer resp)

public void removeResponsibility(Integer resp)

public void removeAllResponsibility()

public int getRefreshInterval()

public void setRefreshInterval(int refreshInterval)

public void load();

public void save();

3.1.3 System Statistics (ST)

SystemStatistics object contains run-time information about the system.

It is used to control the workload in the system for optimal throughput. Data in this object complements the information in the PolicyInformation object. Following information is maintained by the object:

Attribute	Data Type	Description
Depth	int	depth to which currently PRBs are processed
FabricationRate	int	number of PRBs processed per minute
CPUUtilization	double	the current CPU utilization
JservCount	int	total number of Jservs available for processing prefabrication requests

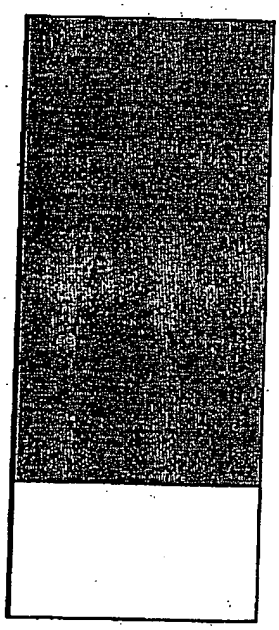
All of the above attributes have *get* and *set* methods to access them.

3.1.4 User Page Table (UPT)

The User Page Table keeps track of information about prefabricated pages that the Benefit Analyzer needs.

Information stored

The User Page Table is a single column table with a list of Page Request Blocks.



Page Request Block (PRB)

The PRB contains information about a particular jsp page. It is populated by Start Loader, Benefit Analyzer and Page Generator.

APIs

The User Page Table will be populated by different components within the system. However, all such writes are executed through APIs exposed by the Benefit Analyzer. Currently writes can be done 1 PRB at a time or in batches. More details can be found in the Benefit Analyzer section. Following is the list of APIs supported by the UserPageTable:

void add(PageRequestBlock)

PageRequestBlock removePRB(String URI)

void remove (PageRequestBlock)

int size()

Vector getAll ()

Vector find (int userID, int appID, int respID)

Vector find (String URISubString)

PageRequestBlock find (String URI)

Vector getPRBOrderBy(DEPTH/WEIGHT, int count, ASC/DESC)

Who populates this table?

Components that would write to the User Page Table are:

Start Loader

During startup, the Start Loader generates all possible URL requests for the homepage. These URL are used to populate the User Page Table.

Page Generator

While generating a page, the Page Generator keeps track of the time spent on prefabricating the page and populates the User Page Table accordingly.

URL Collector

Given the URL of a prefabricated page from the Page Generator (assuming it is at level n), the URL Collector crawls through all the "prefabrication-eligible" URL responses possible from this prefabricated page. This list of URL are used to populate the User Page Table.

Who reads this table?

Benefit Analyzer

The Benefit Analyzer makes use of the information in the User Page Table to determine what pages to prefabricate and their priorities.

Data Structure Requirements

The data structure is still undecided. But the requirements are as follows:

Speed

This table will be accessed very frequently. Given the huge size of the table (the number of prefabricated pages), a very efficiently data structure to retrieve and store data is needed.

Synchronization Needs

Different threads may access the User Page Table at the same time and synchronization is needed to preserve consistency on the table data. Locking of the table is needed when the table is updated. Read only APIs does not need to be synchronized.

Method of Traversal

The Benefit Analyzer needs to access this table to obtain prefabrication information. Three methods of traversal are proposed for the time being.

By Weight

The Benefit Analyzer needs to access all pages with a specified weight.

By User

The Benefit Analyzer needs to access all pages of a particular user.

By URL

The Benefit Analyzer needs to access all pages with a particular url portion. One example is that it needs to access all jsp's of the same name (jtfasud.jsp).

Additional Requirements

- Another method of traversal that might be useful in the future would be to enable a user-defined key (e.g. respID) to scan the table and retrieve pages.
- Yet another possible method of traversal would be to scan the table by specifying the level the pages are at. The usefulness of this feature is to be determined.

3.2 Start Loader

This component does the initial bootstrapping for the prefabricator system. Given the home page (entry point into an application), this component will generate all possible URL requests for the home page. The generated PRBs is populated in the *user page table* so that the benefit analyzer can use it.

Requirements

Given an application's home page (a jsp file name), generate valid URLs for all users who can access the homepage.

Given any application page (a jsp file name), provide mechanism for the application developer to generate valid URL requests for this application page *[Future]*

Implementation Details

In order to provide a generic URL evaluation mechanism, a *PRBEvaluateInterface* is defined. A default implementation of this interface is provided to handle the "home page" URL generation. Application developers can provide their own implementation of the interface to support other application page URL generation. Following is the PRBEvaluateInterface specification:

getPRBs(): constructs a Vector of valid PRB objects

getParameterNames(): returns a String array of URL parameter names that the page expects

The StartLoader implements the Runnable interface. The *run()* method implements the following logic:

It uses the PRBEvaluateInterface implementer to generate all the PRBs that make up the initial workload for the benefit analyzer

Once the PRBs are ready, it invokes the benefit analyzer, to populate the entire batch of PRBs in the user page table.

Periodically, this thread wakes up to check if new PRBs need to be created and creates them. This handles the case where new users register into the system.

The default implementer of the PRBEvaluateInterface class does the following:

query the fnd_user, fnd_user_resp_groups and fnd_responsibility tables to determine all user's of a given application with a specific responsibility - i.e., determine all {userID, appID, respID} combinations for a given application and a set of valid responsibility keys.

construct a PRB object for each valid {userID, appID, respID} combination

3.3 Benefit Analyzer

At any given point of time and state of the prefabricator system, Benefit Analyzer determines the next set of pages to prefabricate. It receives PRBs to be processed both from the "start loader" and the "URL collector".

Selection Algorithm

As stated earlier, the primary goal of the algorithm is to determine the next set of PRBs to be fabricated at any given state of the system. The algorithm needs to meet the following requirements:

At system start-up time, determine initial workload set

Filter (or select) PRBs to be processed based on the current fabrication rate of the system

If fabrication rate of the system < refresh interval specified in the policy table, should definitely process all the user home pages (in other words, highest ranked PRBs)

the algorithm should result in user home pages being always ranked the highest

The algorithm uses the following information that is available to it at all times:

all prefabricated pages (PRBs)

policy information from the policy table

staleness of home pages - indicated by the timestamp of the home page PRBs in the user page table

current prefabrication rate - implicitly determined by the staleness of home pages (or highest ranked PRBs)

The algorithm works as follows:

Anytime a PRB is added to the user page table, it calculates a weight for the PRB using the following formula:

$weight = (1/depth) + k$

where,
depth = depth of the PRB in the tree,
k = some constant to indicate the relevance of the URL to the responsibilities in the policy table
The above formula will ensure that home pages will always have the highest weight, followed by the next depth pages and so on. Furthermore, home pages belonging to users of a responsibility specified in the policy table will have higher weight than those that are not.

Periodically (every 1 minute for instance) the benefit analyzer examines the user page table to issue the next set of PRBs to be processed. The PRBs are selected based on their weight - higher weight PRBs are processed before lower weight ones.

At system start-up time, when current fabrication rate is not available, the benefit analyzer issues PRBs to be processed in batches of a fixed size - 1000 (for example). Also, it considers all the PRBs in the user page table. Over time, the PRBs considered will determine both on the weight and on the staleness of the highest ranked (home page) PRBs. If home page PRBs are stale beyond an acceptable time, it implies that the current fabrication rate is pretty slow. Therefore, the analyzer will focus only on the home pages and makes sure all the home pages get generated at regular intervals.

Implementation Details

The benefit analyzer provides the following services to be used by different components within the system:

Adding PRBs to be prefabricated into the user page table:

addPRB(pageRequestBlock, timeStamp)

addPRB(pageRequestBlock)

addPRB(Vector pageRequestBlocks, Vector timeStamps)

addPRB(Vector pageRequestBlocks)

PRBFilter:

getPRBs(): returns a Vector of PRBs to be processed in a queue. This implements the selection algorithm

getNextPRBs(currentIndex, batchSize): returns a Vector of next set of PRBs to be processed (works like a cursor access)

isPRBInteresting(PRB): returns a boolean to indicate if PRB should be considered for further processing or not

The benefit analyzer executes as a thread that periodically wakes up, selects PRBs to be processed and issues these to the "Request Feeder".

Future & Open Issues

the selection algorithm can be more fine-tuned. One option is to factor in the fanout associated with URLs at different depths. If from a page one can navigate to *n* pages, then each of those n pages have a probability of hit of 1/n (=p). Use this in the weight formula: $w = p*(1/depth) + k$. Another option is to factor in user hit counts in the weight formula.

finalize the APIs associated with this component

3.4 Request Feeder

Request Feeder performs the page generation from the Page Request Blocks.

Algorithm:

The Request Feeder (RF) maintains two internal queues (1) The incoming queue. (2) The outoing queue. The elements in the queue are PRB's.

The Benefit analyzer puts PRB's in the incoming queue for page generation.

The Request Feeder picks elements from the incoming queue. It issues the request for page generation and gets the HTML response stream back. It sets the a reference in PRB to point to the HTML response stream.

The Request Feeder update the generated time in the PRB.

The Request Feeder updates the current page generation rate for the PRB in the SystemStats object.

The generated PRB is put in the outgoing queue. The URL collector picks up the PRB.

Open:

Whether or not the PRF saves the HTML response as a file on the file system or the interceptor does it.

Data Structure:

The Request Feeder maintains two instances of QueueManager class one representing the incoming queue and the other representing the outgoing queue. The QueueManager class internally maintains three priority queues (1) Low (2) Medium (3) High. Each queue is maintained as a vector. The element type of the queue is the Page Request Block.

API's:

QueueManager:

```
public interface QueueManager
{
 public static final int LOW = 0;
 public static final int MEDIUM = 1;
 public static final int HIGH = 2;
/**
 * returns the next QueueElement. The next QueueElement is chosen from
 * amongst the three Queues. The weight of each element is a function of
 * the time spent in the queue and the priority of the queue it is in. The
 * element returned is removed from the corresponding queue.
 * @return the next QueueElement
 * @exception oracle.apps.jtf.base.resources.FrameworkException if an error occurs
 */
 public QueueElement getNextElement()
  throws FrameworkException;
/**
 * returns the next QueueElement. The next QueueElement is chosen from
```

```
     * amongst the three Queues. The weight of each element is a function of
     * the time spent in the queue and the priority of the queue it is in. The
     * element returned is NOT removed from the corresponding queue.
     * @return the next QueueElement
     * @exception oracle.apps.jtf.base.resources.FrameworkException if an error occurs
     */
    public QueueElement peekNextElement()
      throws FrameworkException;
    /**
     * returns the next QueueElement from the specified queue. The
     * element returned is NOT removed from the corresponding queue.
     * @param priority the priority of the queue.
     * @return the next QueueElement
     * @exception oracle.apps.jtf.base.resources.FrameworkException if an error occurs
     */
    public QueueElement peekNextElement(int priority)
      throws FrameworkException;
    /**
     * adds a QueueElement to the specified queue.
     * @param elem the QueueElement to be added.
     * @param priority the priority of the queue.
     * @return the added element
     * @exception oracle.apps.jtf.base.resources.FrameworkException if an error occurs
     */
    public QueueElement addToQueue(Object elem,int priority)
      throws FrameworkException;
    /**
     *  removes a QueueElement from any queue in which it is found
     * @param elem the QueueElement to be added.
     * @param priority the priority of the queue.
     * @return the removed QueueElement
     * @exception oracle.apps.jtf.base.resources.FrameworkException if an error occurs
     */
    public QueueElement removeFromQueue(Object elem)
      throws FrameworkException;
    /**
     *  removes a QueueElement from specified queue
```

```
     * @param elem the QueueElement to be added.
     * @param priority the priority of the queue.
     * @return the removed QueueElement
     * @exception oracle.apps.jtf.base.resources.FrameworkException if an error occurs
     */
    public QueueElement removeFromQueue(Object elem, int priority)
      throws FrameworkException;
    /**
     *  removes the top from specified queue
     * @param priority the priority of the queue.
     * @return the removed QueueElement
     * @exception oracle.apps.jtf.base.resources.FrameworkException if an error occurs
     */
    public QueueElement removeFromQueue(int priority)
      throws FrameworkException;
    /**
     * tells whether or not all the queues are empty
     * return true if all the queues are empty, false otherwise
     * @exception oracle.apps.jtf.base.resources.FrameworkException if an error occurs
     */
    public boolean isEmpty()
      throws FrameworkException;
    /**
     * tells whether or not the specified queue is empty
     * @param  priority the queue priority
     * return true if the specified queue is empty, false otherwise
     * @exception oracle.apps.jtf.base.resources.FrameworkException if an error occurs
     */
    public boolean isEmpty(int priority)
      throws FrameworkException;
    /**
     * tells whether or not all the queues are full
     * return true if all the queues are full, false otherwise
     * @exception oracle.apps.jtf.base.resources.FrameworkException if an error occurs
     */
    public boolean isFull()
      throws FrameworkException;
```

```
/**
 * tells whether or not the specified queue is full
 * @param  priority the queue priority
 * return true if the specified queue is full, false otherwise
 * @exception oracle.apps.jtf.base.resources.FrameworkException if an error occurs
 */
public boolean isFull(int priority)
  throws FrameworkException;
/**
 * returns the size of the specified queue
 * @param priority the queue priority
 * @return the number of elements in the queue
 * @exception oracle.apps.jtf.base.resources.FrameworkException if an error occurs
 */
public int getQueueSize(int priority)
  throws FrameworkException;
/**
 * returns the total size of all the queue
 * @return the number of elements in the queue
 * @exception oracle.apps.jtf.base.resources.FrameworkException if an error occurs
 */
public int getQueueSize()
  throws FrameworkException;
/**
 * returns the maximum queue size allowed for a queue
 * @param priority the priority of the queue
 * @return the maximum queue size allowed
 * @exception oracle.apps.jtf.base.resources.FrameworkException if an error occurs
 */
public int getMaxQueueSize(int priority)
  throws FrameworkException;
/**
 * returns the maximum queue size allowed for all the queues
 * @return the maximum queue size allowed
 * @exception oracle.apps.jtf.base.resources.FrameworkException if an error occurs
 */
public int getMaxQueueSize()
```

```
    throws FrameworkException;
  /**
   * set the maximum queue size of the given queue
   * @param priority the priority of the queue
   * @exception oracle.apps.jtf.base.resources.FrameworkException if an error occurs
   */
  public void setMaxQueueSize(int priority)
   throws FrameworkException;

}
```

QueueElement:

```
public class QueueElement {

 private Object element_;
 private int priority_;
 private long timestamp_;

 public QueueElement(Object element, int priority);

 public Object getElement();
 public int calculateRank();

 public int compare(QueueElement qe);
}
```

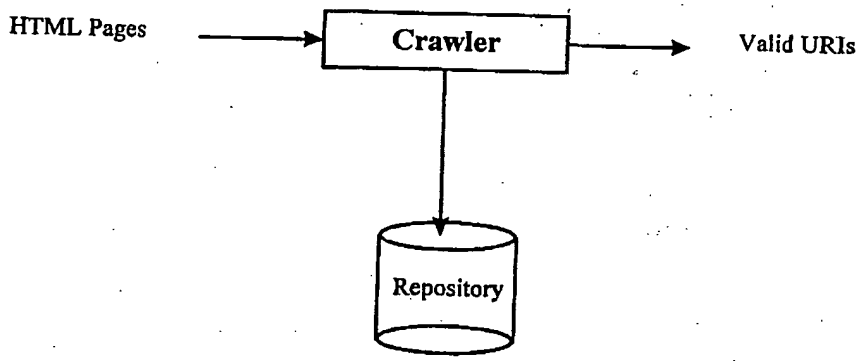
PageRequestFeeder:

```
public class PageRequestFeeder
{
 public PRB getElement()
 throws FrameworkException;
 public boolean putElement(PRB pr)
 throws FrameworkException;
 public boolean generatePRB(PRB pr)
 throws FrameworkException;
```

```
public float getPRBRate()
throws FrameworkException;
}
```

3.5 URL Collector

The "URL Collector" component functions like a web-crawler. It saves HTML pages to the cache and also walks through them to produce sets of valid URIs (Unique Resource Identifier) that are eligible for pre-fabrication.



To minimize the possibility of duplicates/circular-references the URL Collector will generate and store a unique HashId for each URI it generates. Each time a URI is generated, it's HashId will be compared to the existing list of existing HashIds. If a match is found the URI will be discarded with the reasonable assumption that it has already been generated in the past.

The same unique HashId will also be used by the Interceptor for looking up cached pre-fabricated pages.

3.6 Page Generator

The Page Generator will receive requests for pre-fabrication, communicate the request to the JServ/Web-Server, and feed the processed HTML output to the Crawler module.

Open Issue
Need to decide how the URL Collector will get the responses for the user requests. Can be a HTTPClient which is sent a response by the "interceptor".

3.7 Page Invalidator

Three types of invalidation schemes are supported:

- interval-based invalidation: requests for re-generating a particular URL are issued at regular intervals. The

length of the intervals is dependent on the resources (hardware & database), frequency of updates and consistency level. This is handled by the benefit analyzer.

- event-based invalidation: when an update request is issued, all the dependent URLs (if available) get invalidated. In order to use this invalidation scheme, the application developer would have to register the following information:
  - list of updateable JSP pages: jsp page name, URL parameters (if any) that make it updateable
  - dependencies: it can be a list of JSP pages, or list of parameter values for user-defined URL key. Common case is user ID. E.g., when a report is inserted by an user, all pages for this user should be invalidated.
- user-login invalidation: to minimize the inconsistency window, the moment an user logs in, URLs for this user start getting re-generated.

Open Issues

1. For event based invalidation, the Interceptor would have to communicate the update request that was executed to the invalidator component, and to all other interceptors
   - more details on this issue are addressed in the following interceptor section
   - if Calypso is used, it would handle this
2. For user-login based invalidation, the interceptor has to communicate to the invalidator:
   - more details on this issue are addressed in the following interceptor section

3.8 User Session Management

User session management has to be done at two levels:

- when prefabricator issues requests for generation, requests need to belong to valid user sessions
- when a user transitions from a prefabricated page to a non-prefabricated page, user session should not change

This section describes how the above two cases are handled. In order to support user session management for the prefabricator requests, a special starting page for all applications is created - *jtfprefab.jsp*. This page replaced the normal *jtfavald.jsp*. It does the following:

1. Takes userID, appID and respID as inputs
2. Authenticates a special prefabUser
3. Switches user session context to that specified by userID - invokes setUser
4. Sets up appID and respID as the values of application and responsibility ID for this session and determines the entry page (home page) for this set of values.
5. Sends back response with complete URL for the home page calculated above along with sessionID.

When an user request is received, the interceptor sets the user's sessionID to be the same as that created for the prefabricated pages belonging to this user. This ensures smooth session transition between prefabricated and non-prefabricated pages.

3.9 Interceptor

This component intercepts the URL requests, determines if the request can be serviced from the cache or

not. The basic algorithm implemented by the interceptor is as follows:

Check if request is from prefabricator or from user (by examining a parameter in the request object):

If from user:

Check if request exists in the cache. If it does, modify the ssoCookie to prefabricator generated ssoCookie and return the response.

If not present in cache, service the page

If request is an update request, mark all dependent pages as invalid. Dependent pages are stored in a static "Page Registration" table.

If from spider:

Service the page

Store response in the cache

In either case, response should also be T-d to the URL Collector component of the prefabricator.

Open Issues

How efficiently can the response-T be implemented?

When there are multiple apache hosts, how does synchronization of invalidation information get propagated?
- could range-partition requests by userID across different apache hosts. This will maintain user-level strict consistency, but not strict consistency across multiple users. E.g.: sales representative updates a report, manager would want to see it. But how much strict consistency is actually needed?

Within a single apache host, how does synchronization of invalidation information across multiple Jservs work?

- if implemented in C, can use shared-memory
- plug-in our own "load-balancing" algorithm into apache, so that requests from a range of users is always sent to a specific Jserv. i.e., range-partition by userID across Jservs

Need clear design of ssoCookie update mechanism

3.10 Cache

The cache provider needs to support the following:

store response in the cache based on a key

get response from the cache based on a key

by-pass the cache, and actually hit the server on-demand (specifically needed from Calypso which is a cache and reverse-proxy)

modify response object with interceptor specified information (such as cookies) if the cache is handling the returning of the response (specifically needed from Calypso which is a cache and reverse-proxy)

Open Issues: Calypso Requirements

pass-through to service the request, rather than using the cache

modifyResponseHeader(..) - ability to modify the response header before sending to client

4 Internal APIs/Classes

If the product will have a collection of internal function calls, they should all be listed here (reference the header file), along with:

- the number of round trips
- a description of what the function does
- the input and output parameters and descriptions
  - Pass by ref, value, pointers
  - null pointers expected to be initialized, or Ok to return empty
  - memory allocation/deallocation, e.g. if client allocates, they deallocate

If the product includes classes, it should include similar information about the classes, how they relate to each other (preferably with a class hierarchy diagram), and all of the attributes and methods of each class.

Similar information is to be provided for OCI, PRO, and Java interfaces.

B Glossary

If your project uses a lot of terminology that may be new to your reviewers, you should consider adding a glossary section. Otherwise, just describe it in the Concepts section.

<term>

<definition>

Examples:

Gizmo

A peculiar construct.

Widget

A widget is an instance of a gizmo.

D Open Issues

Include this section in all review copies of the specification. List all open issues and identified risks with the design.